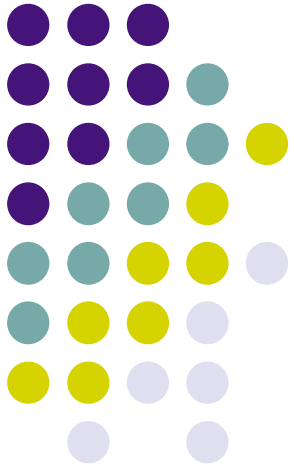
Extensions to I/O

Tom Clune

SIVO Fortran 2003 Series

March 11, 2008



Logistics

- Materials for this series can be found at http://modelingguru.nasa.gov/clearspace/docs/DOC-1375
  - Contains slides and source code examples.
  - Latest materials may only be ready at-the-last-minute.
- Please be courteous:
  - Remote attendees should use “*6” to toggle the mute. This will minimize background noise for other attendees.
- Webex - under investigation

Outline

- Major extensions
  - Stream I/O
  - Asynchronous
  - Derived Type I/O
- Miscellaneous
  - Recursive I/O
  - Named constants: `ISO_FORTRAN_ENV`
  - New statements/intrinsics: `FLUSH(), NEW_LINE()`
  - New optional keywords
    - `IOMSG`
    - `SIGN`
    - `DECIMAL`
    - `ROUND`
  - Miscellaneous, miscellaneous
- Pitfalls and Best Practices
- Resources

Stream I/O

- Stream access is a new method for allowing fine-grained, *random* positioning within a file for read/write operations.
  - Complements pre-existing DIRECT and SEQUENTIAL access
  - Advantages of STREAM access:
    - Random access (as with DIRECT)
    - Arbitrary record lengths (as with SEQUENTIAL)
    - No vendor dependent record separators (as with DIRECT), which enables both portability and interoperability with other languages
  - Disadvantages of STREAM access:
    - Presumably poorer performance than both DIRECT and SEQUENTIAL
    - Lack of record separators increases risk of inability to read file under small changes.
    - Index for positioning within file might be less natural than those for DIRECT.
- To open a file for stream I/O use `ACCESS='STREAM':`

```
OPEN(unit, ACCESS = 'STREAM')
```

  - Both formatted and unformatted I/O are supported

Stream I/O (cont'd)

- Read/write to stream file use `POS` keyword to specify position:

```
READ(unit,POS=n) x,y,z
```

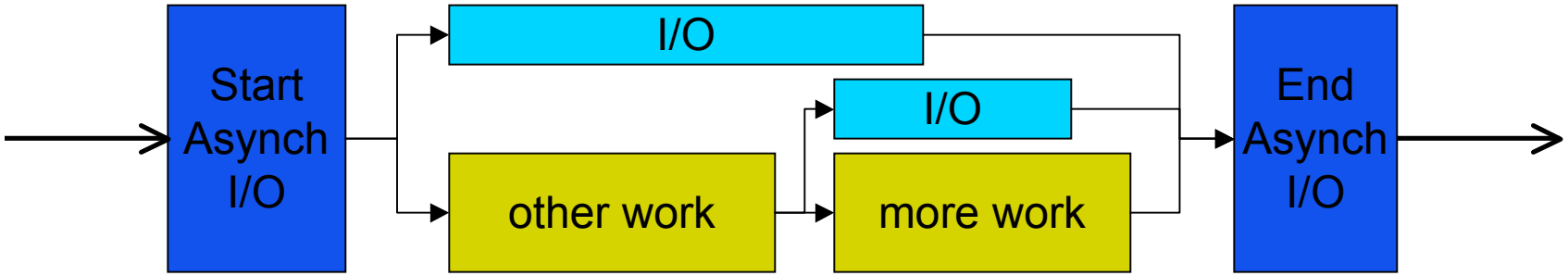
  - File starts at position `POS=1` (not zero!)
  - Position is specified in "file storage units" - usually bytes
    - Useful constant- `ISO_FORTRAN_ENV::FILE_STORAGE_SIZE`
  - If POS keyword is omitted, access continues from last access.
- `INQUIRE()` uses **`POS`** keyword to retrieve current position

```
INQUIRE(unit, POS=currentPosition, …)
```

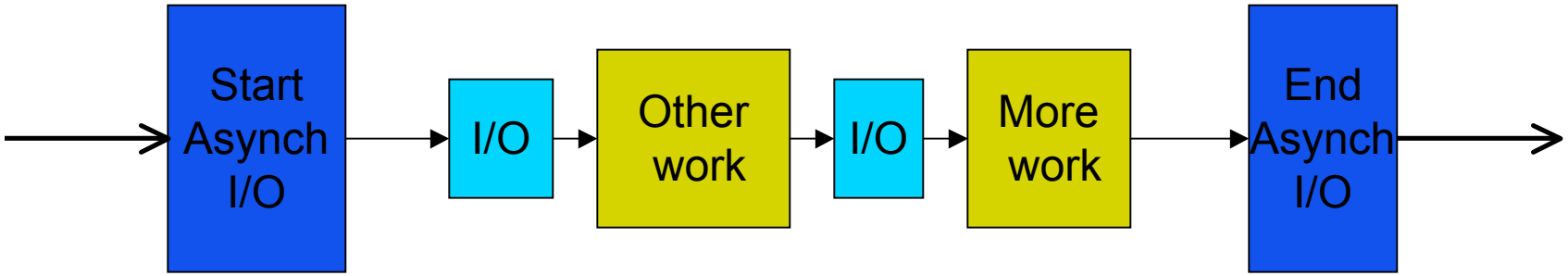
- Restrictions:
  - Formatted I/O must use `POS` obtained from `INQUIRE()` (or `POS=1`)
  - Vendors may prohibit `POS` for certain file types

Asynchronous I/O

- Potential performance enhancement allowing some I/O operations to be performed in parallel with other computations.
  - Multiple I/O operations may progress simultaneously.



- Note that the standard allows vendors to implement with completely synchronous operations.



Asynchronous I/O cont'd

- To open a file for asynchronous operations, the new optional keyword `ASYNCHRONOUS` is used

```
open(unit, 'file', ASYNCHRONOUS='yes', …)
```

- An asynchronous read/write operation is initiated with the same keyword:

```
write(unit,ASYNCHRONOUS='yes') …
```

  - Data items in the I/O list are referred to as '*affectors*'.
  - Operation itself is referred to as '*pending*'.
  - Note that the default is `ASYNCHRONOUS='no'` even if the file was opened with 'yes'.

- An optional keyword, ID, can be used to return a handle for later use in identifying specific pending operations:

```
integer :: asyncID
read(unit,…,ID=asyncID)
```

Asynchronous I/O

- Pending operations are terminated by any “wait” operation
  - Explicit wait statement: `WAIT(unit)`
  - Implicit wait via `CLOSE()`, `INQUIRE()`, or file positioning statement
- New optional keyword **`PENDING`** for `INQUIRE()` statement
  - Returns logical scalar indicating whether operation has completed.

```
logical :: isPending
INQUIRE(unit, PENDING=isPending)
if (isPending) …
```

- Both WAIT() and INQUIRE() statements accept the optional keyword “ID” to specify specific I/O operations:

```
write(unitA, ID=idA) bigArrayA
write(unitB, ID=idB) smallArrayB
INQUIRE(unit, ID=idA, PENDING=isPending) !1st write
WAIT(unit,ID=idB) ! Only wait for second write()
```

Asynchronous I/O cont'd

- Certain restrictions required to guarantee *consistency* during pending operations:
  - Output affectors may not be *modified* during pending operations
  - Input affectors may not be *referenced* at all during pending operations
- Affectors may be declared with `ASYNCHRONOUS` attribute

```
REAL, ASYNCHRONOUS :: array(IM,JM,LM)
REAL :: otherArray(N)
ASYNCHRONOUS :: otherArray
```

  - Warns a compiler that certain optimizations may be prohibited
  - *Automatic* for affectors in the *scoping unit*.
  - Needs to be explicit for any variable which is an affector in another scoping unit.
    - Dummy variables and variables accessed by host association.
    - Attribute can be specified without *redeclaring* variable:

```
ASYNCHRONOUS :: varFromOtherModule
```

Derived Type I/O

- Standard allows for user-defined I/O of derived type
  - When derived type is encountered in an I/O list, a Fortran subroutine is called.
    - Reads some data and constructs a value of the derived type *or*
    - Writes some data from a derived type into a file
  - Support for both FORMATTED and UNFORMATTED
    - Formatted I/O edit descriptor an extra string and integer array that can be used to control operations
    - Example FORMATTED edit descriptor:

```
DT 'linked-list' (10, -4, 2)
```

      - If string is omitted it is treated as string of length 0
      - If Array is omitted, it is treated as an array of size 0

Derived Type I/O (cont'd)

- Two mechanisms are provided to associate a subroutine with I/O for a derived type
  1. So-called type-bound procedures - deferred until OO
  2. Interface block

```
INTERFACE READ(FORMATTED)
      module procedure readType
END INTERFACE
```

Derived Type I/O (cont'd)

- Derived type I/O subroutines must conform to a very specific interface:

```
SUBROUTINE formatted_io  (dtv,unit,iotype,v_list,iostat,iomsg)
SUBROUTINE unformatted_io(dtv,unit,                iostat,iomsg)
```

  - `DTV` is a scalar of the derived type
    - `Intent(IN)` for write operations
    - `Intent(INOUT)` for read operations
  - `UNIT` is a default integer of `intent(IN)`
    - Negative for internal file
  - `IOSTAT` is an intent(out) default integer
    - Must be given positive value on error
    - Enf-of-file or end-of-record must be set to `IOSTAT_END` or `IOSTAT_EOR` respectively
  - `IOMSG` character(*), intent(inout)
    - If `IOSTAT` is positive, `IOMSG` must be given an explanatory message.

Derived Type I/O (cont'd)

- Formatted I/O has two additional mandatory arguments.
  - These provide additional flexibility for altering format of I/O depending on context
  - `IOTYPE` is `character(*), intent(in)`
    - Value depends on context of actual I/O operation
    - '`LISTDIRECTED`'
    - '`NAMELIST`'
    - '`DT`'//*string* where *string* is from the DT edit descriptor.
  - `VLIST` is an intent(in), rank-1 integer array of assumed size from the edit descriptor

Derived Type I/O (cont'd)

- Some caveats:
  - Input/Output operations in these subroutines are limited to the specified unit and the specified direction (read/write).
    - However, operations to internal files are permitted
  - The file position on entry is treated as a left tab limit and there is no record termination on return.
  - Derived type I/O is not available in combination with asynchronous input/output.

Miscellaneous

- Recursive I/O
- Named constants: `ISO_FORTRAN_ENV`
- New statements/intrinsics: `FLUSH(), NEW_LINE()`
- New optional keywords
  - `IOMSG`
  - `SIGN`
  - `DECIMAL`
  - `ROUND`
- Miscellaneous, miscellaneous

Recursive I/O

- Previous versions of the standard prohibited all recursive I/O operations due to ambiguity about expected results
- New standard relaxes these restrictions in the special case of *internal* files:

```
function toString(n) result(string)
   integer, intent(in) :: n
   character(len=3) :: string
   write(string,'(i3.3)') n
end function toString
…
write(unit,*) toString(i), toString(j), toString(k)
```

ISO_FORTRAN_ENV

- Intrinsic module for named I/O constants - portability
- Standard units - default integer scalars:
  - `INPUT_UNIT` - unit '*' in `READ` statement
  - `OUTPUT_UNIT` - unit '*' in `WRITE` statement
  - `ERROR_UNIT` - used for error reporting
- Vendor dependent integer scalars with values that are assigned to `IOSTAT=` if an end-of-file or end-of-record condition
  - `IOSTAT_END`
  - `IOSTAT_EOR`
- Size in *bits* for numeric, character and file storage:
  - Supports portability
  - `NUMERIC_STORAGE_SIZE`
  - `CHARACTER_STORAGE_SIZE`
  - `FILE_STORAGE_SIZE`

New statement and intrinsic

- New statements
  - `WAIT()` - saw this in asynchronous I/O
  - `FLUSH(unit)`
    - Makes *written* data available to other processes
    - Makes data from other processes available to *read*
    - With `ADVANCE='NO'` or stream access, permits access to keyboard input character-by-character
- New Intrinsic
  - `NEW_LINE(A)`
    - Function which returns a ‘newline’ character
    - ‘A’ is of type character and specifies the KIND of the result

Miscellaneous Keywords

- Informative error messages: `IOMSG`
  - Optional keyword to any input/output statement
  - Identifies a scalar variable of default character into which the vendor places a message if an error is encountered.
  - Actual argument is unchanged if there is no error
  - Actual message is vendor dependent.
- Optional '+' in formatted numeric output: `SIGN`
  - Sets file default in `OPEN()`
  - Override in `WRITE()` statement with `SS`, `SP` and `S` edit descriptors
  - Allowed values: `SUPPRESS`,`PLUS`, & `PROCESSOR_DEFINED`

Keywords (cont'd)

- Portability with *Europeans*: `DECIMAL`
  - Controls the character that separates the parts of a decimal number in *formatted* I/O
  - Default set with `open()`

```
open(unit, …,DECIMAL=<specifier>,…)
```

  - Allowed values are `COMMA` or `POINT`
  - Can override default for file in read/write statements with 'DC' and 'DP' edit descriptors.

Keywords (cont'd)

- Rounding during formatted input/output: `ROUND`
  - Set default in OPEN() statement

```
open(unit, …, ROUND=<specifier>,…)
```

  - Permitted values:
    - `UP`
    - `DOWN`
    - `ZERO`
    - Closest value:
      - `NEAREST` - processor dependent if equidistant
      - `COMPATIBLE` - away from zero if equidistant
    - `PROCESSOR_DEFINED`
  - Can be locally overridden in READ/WRITE statements by `RU`, `RD`, `RZ`, `RN`, `RC`, and `RP` edit descriptors

Miscellaneous miscellaneous

- Input and output of IEEE infinities and NaNs
  - Unconstrained in F95 and earlier
  - Uses edit descriptors for reals - only width ‘W’ is taken into account
  - Output forms are
    1. `-Inf` or `-Infinity` for minus infinity
    2. `Inf`, `+Inf`, `Infinity`, or +Infinity for plus infinity
    3. `NaN`, optionally followed by non-blank characters in parentheses
  - Each is right justified in its field.
- Any kind of integer is permitted for I/O keywords
  - Default integers are just too small for some applications.
- Comma after ‘P’ edit descriptor is optional when followed by a repeat specifier
  - E.g. `1P2E12.4` is permitted

Pitfalls and Best Practices

- `ASYNCHRONOUS`
  - Watch for race conditions
  - Declare with ASYNCHRONOUS in other scoping units
- `STREAM`
  - Use `INQUIRE` to obtain `POS` in file (avoid formulae)
  - File storage size from `ISO_FORTRAN_ENV`
- Use IOMSG to obtain informative error messages
- Use named constants when possible

Supported Features

Compiler	Ifort 9.1.049	Ifort 10.1	NAG 5.1	Xlf 11.0	G95 0.90	Gfortran 20070810	pgi 6.2.4
Asynchronous	no	yes	no	yes	no	no	no
Stream	no	yes	yes	yes	yes	yes	no
Recursive	yes	yes	no	yes	yes	yes	yes
Intrinsic NEW_LINE	yes	yes	yes		yes	yes	no
Derived Type							

Feel free to contribute if you have access to other compilers not mentioned!

Resources

- **SIVO Fortran 2003 series:** https://modelingguru.nasa.gov/clearspace/docs/DOC-1390
- **Questions to Modeling Guru:** https://modelingguru.nasa.gov
- **SIVO code examples on Modeling Guru**
- **Fortran 2003 standard:** http://www.open-std.org/jtc1/sc22/open/n3661.pdf
- ***John Reid summary:***
  - ftp://ftp.nag.co.uk/sc22wg5/N1551-N1600/N1579.pdf
  - ftp://ftp.nag.co.uk/sc22wg5/N1551-N1600/N1579.ps.gz
- ***Newsgroups***
  - http://groups.google.com/group/comp.lang.fortran

Next Fortran 2003 Session

- Miscellaneous
- Tom Clune will present
- Tuesday, March 25, 2008
- B28-E210